Events, Co-routines, Continuations and Threads
OS (and application) Execution Models

System Building

General purpose systems need to deal with

- Many activities
  - potentially overlapping
  - may be interdependent
    - need to resume after something else happens
- Activities that depend on external phenomena
  - may requiring waiting for completion (e.g. disk read)
  - reacting to external triggers (e.g. interrupts)

Need a systematic approach to system structuring

Construction Approaches

Events
Coroutines
Threads
Continuations

Events

External entities generate (post) events.
- keyboard presses, mouse clicks, system calls

Event loop waits for events and calls an appropriate event handler.
- common paradigm for GUIs

Event handler is a function that runs until completion and returns to the event loop.

Event Model

The event model only requires a single stack
- All event handlers must return to the event loop
  - No nesting
  - No yielding

No preemption of handlers
- Handlers generally short lived

What is ‘a’?

int a; /* global */

int func()
{
    a = 1;
    if (a == 1) {
        a = 2;
    }
    return a; // No concurrency issues within a handler
}
Huh? How to support multiple processes?

Co-routines

Originally described in:

Analogous to a "subroutine" with extra entry and exit points.

Via yield():
- Supports long running subroutines
- Can implement sync primitives that wait for a condition to be true
  - while (condition != true) yield();

What is ‘a’?

```c
int a; /* global */

int func() {
    a = 1;
    yield();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

Limited concurrency issues/races as globals are exclusive between yields()
Co-routines Implementation strategy?

Usually implemented with a stack per routine
Preserves current state of execution of the routine

Co-routines

Routine A state currently loaded
Routine B state stored on stack
Routine switch from A → B
- saving state of A
  - regs, sp, pc
- restoring the state of B
  - regs, sp, pc

A hypothetical yield()

```c
yield:
    /*
    * a0 contains a pointer to the previous routine’s struct.
    * a1 contains a pointer to the new routine’s struct.
    *
    * The registers get saved on the stack, namely:
    *
    *     s0-s8
    *     gp, ra
    *
    */
    /* Allocate stack space for saving 11 registers. 11*4 = 44 */
    addi sp, sp, -44
    /* Save the registers */
    sw ra, 40(sp)
    sw gp, 36(sp)
    sw s8, 32(sp)
    sw s7, 28(sp)
    sw s6, 24(sp)
    sw s5, 20(sp)
    sw s4, 16(sp)
    sw s3, 12(sp)
    sw s2, 8(sp)
    sw s1, 4(sp)
    sw s0, 0(sp)
    /* Store the old stack pointer in the old pcb */
    sw sp, 0(a0)
```

```c
Save the registers that the 'C'
procedure calling
convention expects
preserved
```

```c
/* Get the new stack pointer from the new pcb */
lw sp, 0(a1)
/* delay slot for load */
/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s7, 28(sp)
lw s8, 32(sp)
lw gp, 36(sp)
lw ra, 40(sp)
/* delay slot for load */
/* and return. */
jr ra
addi sp, sp, 44 /* in delay slot */
.end mips_switch
```

Yield

Routine A

Routine B

Yield
What is ‘a’?

```c
int a; /* global */

int func() {
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

Coroutines

What about subroutines combined with coroutines
- i.e. what is the issue with calling subroutines?
  Subroutine calling might involve an implicit yield()
- potentially creates a race on globals
  - either understand where all yields lie, or
  - cooperative multithreading

Cooperative Multithreading

Also called green threads
Conservatively assumes a multithreading model
- i.e. uses synchronisation (locks) to avoid races,
  - and makes no assumption about subroutine behaviour
  - Everything thing can potentially yield()

A Thread

Thread attributes
- processor related
  - memory
  - program counter
  - stack pointer
  - registers (and status)
- OS/package related
  - state (running/blocked)
  - identity
  - scheduler (queues, priority)
  - etc...

Thread Control Block

To support more than a single thread we need to store thread state and attributes
- Stored in per-thread thread control block
  - also indirectly in stack
Thread A and Thread B

Thread A state currently loaded
Thread B state stored in TCB B
Thread switch from A → B
  • saving state of thread A
    - regs, sp, pc
  • restoring state of thread B
    - regs, sp, pc
Note: registers and PC can be stored on the stack, and only SP stored in TCB

Approximate OS

OS/161 mips_switch

Save the registers */

Thread Switch

mips_switch(a,b)
  {
    mips_switch(b,a)
  }

mips_switch(a,b)
  {
/* Get the new stack pointer from the new pcb */
lw sp, 8(sp)

/* Now, restore the registers */
lw s8, 32(sp)
lw s7, 28(sp)
lw s6, 24(sp)
lw s5, 20(sp)
lw s4, 16(sp)
lw s3, 12(sp)
lw s2, 8(sp)
lw s1, 4(sp)
lw s0, 0(sp)
sw a0, 0(sp)
addi sp, sp, 44 /* in delay slot */
/* and return. */
j ra
addi sp, sp, 44 /* in delay slot */
.end mips_switch

Preemptive Multithreading

Switch can be triggered by asynchronous external event
- timer interrupt
- on current stack, if in kernel (nesting)
- on kernel stack or in TCB if coming from user-level
  call thread_switch()

Threads on simple CPU

Threads on CPU with protection

Switching Address Spaces on Thread Switch = Processes
What is this?

User-level Threads

- Fast thread management (creation, deletion, switching, synchronisation…)
- Blocking blocks all threads in a process
  - Syscalls
  - Page faults
- No thread-level parallelism on multiprocessor

Kernel-level Threads

- Slow thread management (creation, deletion, switching, synchronisation…)
  - System calls
- Blocking blocks only the appropriate thread in a process
- Thread-level parallelism on multiprocessor

Continuations (in Functional Languages)

Definition of a Continuation
- Representation of an instance of a computation at a point in time
call/cc in Scheme

call/cc = call-with-current-continuation

A function:
- takes a function (f) to call as an argument
- calls that function with a reference to current continuation (cont) as an argument
- when cont is later called, the continuation is restored.
  - The argument to cont is returned from to the caller of call/cc

Note

For C-programmers, call/cc is effectively saving stack, and PC

Simple Example

(define (f arg)
  (arg 2)
)

(display (f (lambda (x) x))); displays 3
(display (call-with-current-continuation f)) ; displays 2

Another Simple Example

(define (text)
  (let ((i 0))
    ; call/cc calls its first function argument, passing
    ; a continuation variable representing this point in
    ; the program as the argument to that function.
    ; In this case, the function argument assigns that
    ; continuation to the variable the-continuation.
    (call/cc (lambda (k) (set! the-continuation k)))
    ; The next time the-continuation is called, we start here.
    (set! i (+ i 1))
    i))

Another Simple Example

> (text)
  1
> (the-continuation)
  2
> (the-continuation)
  1
> ; stores the current continuation (which will print 4 next) away
> (define another-continuation the-continuation)
> (text) ; resets the continuation
  2
> (the-continuation)
  2
> (another-continuation) ; uses the previously stored continuation
  4
Yet Another Simple Example

```scheme
;;; Return the first element in LST for which WANTED? returns a true value.
(define (search wanted? lst)
  (call/cc (lambda (arg)
    (for-each (lambda (element)
      (if (wanted? element)
        (arg element))
      lst)
    #f)))
```

Coroutine Example

```scheme
;;; This starts a new routine running (proc).
(define (fork proc)
  (call/cc (lambda (k)
    (enqueue k)
    (proc))))

;;; This yields the processor to another routine, if there is one.
(define (yield)
  (call/cc
    (lambda (k)
      (enqueue k)
      ((dequeue)))))
```

Continuations

A method to snapshot current (stack) state and return to the computation in the future in the general case, as many times as we like.
Variations and language environments (e.g. in C) result in less general continuations
- e.g. one shot continuations, setjmp()/longjump()
Single Kernel Stack
“Event” or “Interrupt” Model

How do we use a single kernel stack to support many threads?

- Issue: How are system calls that block handled?
  ⇒ either continuations
    - Using Continuations to Implement Thread Management and Communication in Operating Systems. [Draves et al., 1991]
  ⇒ or stateless kernel (event model)
    - Interface and Execution Models in the Fluke Kernel. [Ford et al., 1999]
    - Also seL4

Stateless Kernel
System calls can not block within the kernel

- If syscall must block (resource unavailable)
  - Modify user-state such that syscall is restarted when resources become available
  - Stack context is discarded (functions all return)

Preemption within kernel difficult to achieve.
⇒ Must (partially) roll syscall back to a restart point
Avoid page faults within kernel code
⇒ Syscall arguments in registers
  - Page fault during roll-back to restart (due to a page fault) is fatal.

IPC implementation examples – Per thread stack

<table>
<thead>
<tr>
<th>msg_send_rcv(msg, option, send_size, rcv_size, ...) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = msg_send(msg, option, send_size, ...);</td>
</tr>
<tr>
<td>if (rc != SUCCESS)</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.msg = msg;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.option = option;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.rcv_size = rcv_size;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>msg_rcv_continue() {</td>
</tr>
<tr>
<td>msg = cur_thread-&gt;continuation.msg;</td>
</tr>
<tr>
<td>option = cur_thread-&gt;continuation.option;</td>
</tr>
<tr>
<td>rcv_size = cur_thread-&gt;continuation.rcv_size;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The function to continue with if blocked

IPC examples - Continuations

<table>
<thead>
<tr>
<th>msg_send(msg, option, send_size, rcv_size, ...) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = msg_send(msg, option, send_size, ...);</td>
</tr>
<tr>
<td>if (rc != SUCCESS)</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.msg = msg;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.option = option;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.rcv_size = rcv_size;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

IPC Examples – stateless kernel

<table>
<thead>
<tr>
<th>msg_send(msg, option, send_size, rcv_size, ...) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = msg_send(msg, option, send_size, ...);</td>
</tr>
<tr>
<td>if (rc != SUCCESS)</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.msg = msg;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.option = option;</td>
</tr>
<tr>
<td>cur_thread-&gt;continuation.rcv_size = rcv_size;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);</td>
</tr>
<tr>
<td>return rc;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Set user-level PC to restart msg_rcv only</td>
</tr>
<tr>
<td>RESCHEDULE changes curthread on exiting the kernel</td>
</tr>
<tr>
<td>Send and Receive system call implemented by a non-blocking send part and a blocking receive part.</td>
</tr>
</tbody>
</table>

Continuations
State required to resume a blocked thread is explicitly saved in a TCB
- A function pointer
- Variables
Stack can be discarded and reused to support new thread
Resuming involves discarding current stack, restoring the continuation, and continuing

example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    save_arg_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P1();
  }
  thread_syscall_return(SUCCESS);
}

example_continue() {
  recover_arg_from_TCB;
  P1(recovered_arg);
  thread_syscall_return(SUCCESS);
}
Single Kernel Stack
per Processor, event model

either continuations
- complex to program
- must be conservative in state saved (any state that might be needed)
- Mach (Draves), L4Ka:Strawberry, NICTA Pistachio, OKL4

or stateless kernel
- no kernel threads, kernel not interruptible, difficult to program
- request all potentially required resources prior to execution
- blocking syscalls must always be re-startable
- Processor-provided stack management can get in the way
- system calls need to be kept simple “atomic”.
  e.g. the fluke kernel from Utah

low cache footprint
- always the same stack is used
- reduced memory footprint

Per-Thread Kernel Stack

simple, flexible
- kernel can always use threads, no special techniques required for keeping state while interrupted / blocked
- no conceptual difference between kernel mode and user mode
  e.g. traditional L4, Linux, Windows, OS/161

but larger cache footprint
and larger memory consumption